

## 第6章 モナドと命令型言語の意味

IO は効率のために Haskell の処理系で特別扱いされる組込みのモナドであるが、他の言語の副作用を模倣するためにユーザーがモナドを定義することも可能である。モナドを用いる利点は、“計算”の意味が変わっても、モナドの標準的な関数 *return* と  $(\gg=)$  のみを用いている部分は、変更する必要がないところである。

以下では、簡単な命令型プログラミング言語（つまり副作用を持つ言語）を定義し、モナドを利用してその意味を Haskell で与えることにする。Haskell で意味を与えるということは、結局はラムダ計算で意味を与えることになる。（等式による推論が可能になる。）具体的には命令型プログラミング言語から Haskell へのコンパイラを作成する。

### 6.1 Util コンパイラ

この節では、コンパイラの実装を紹介していく。実装の詳細は把握しなくても、ソースプログラムとターゲットプログラムを比べれば、副作用を持つプログラミング言語がどのように変換されるか感覚的に掴めるはずである。

簡単な言語からはじめて様々な特徴をもつ言語を定義していく。名前がないと不便なので、これらの命令型言語を Util (Util: Tiny Imperative Language)<sup>1</sup> と呼び、必要により、UtilErr, UtilST, UtilCont, ... などのようにバージョンを表す接尾語をつけることにする。いろいろな特徴を導入していくにつれ、その“計算”を表すモナドの定義が変わることになる。

実際のコンパイラにはフロントエンド、つまり \_\_\_\_\_ (空欄 6.1.1) や \_\_\_\_\_ (空欄 6.1.2) が必要である。字句解析や構文解析の原理は Haskell でも C 言語などの命令型言語で記述するときと変わりはない。再帰下降構文解析法（あるいは LR 構文解析法）などの方法を利用する。（ただし、再帰下降法で構文解析部を記述するときに、後述のようにモナドを利用することができる。）

---

しかし、ここではこれらフロントエンドの作り方は既知のものとして、構文木ができた状態から話をはじめることにする。

---

<sup>1</sup>いわゆる再帰的頭字語 (recursive acronym) である。PHP, GNU などの略語の由来も参照すること。

### 6.1.1 構文規則

Util の構文木のデータ構造として、次のような Haskell データ型を使用する。  
ファイル RecType.hs

```
1  type Decl = (String, Expr)
2  data Expr = Const Target          -- 定数 (Target は後述)
3             | Var String           -- 変数
4             | If Expr Expr Expr    -- if 文
5             | While Expr Expr      -- while 文
6             | Begin [Expr]         -- ブロック
7             | Let [Decl] Expr      -- let 式 (関数定義)
8             | Val Decl Expr        -- val 式 (変数定義)
9             | Lambda String Expr   -- ラムダ式
10            | Delay Expr           -- delay 式 (後述)
11            | App Expr Expr        -- 関数適用
12            deriving Show
```

つまり、式 (Expr) とは、定数 (Const) または、変数 (Var) または、if 式 (If)、let 式 (Let)、ラムダ式 (Lambda)、関数適用 (App) などからなる。(あとから必要に応じて構文要素を追加することにする。)

Util の具体的な構文としては次のような BNF で定義されていると仮定する。(演算子の優先順位なども適切に宣言されているとする。)

```
Expr  →  Const | Var | ( Expr )
       |  if Expr then Expr else Expr | while Expr do Expr
       |  begin Exprs end
       |  let Decls in Expr | val Decl in Expr
       |  \ Var -> Expr | Expr Expr
       |  Expr+Expr | Expr*Expr | … ( 他の中置演算子 ) …
Exprs →  Expr | Expr ; Exprs
Decl  →  Var=Expr
Decl  →  Decl | Decl ; Decls
```

ここに示されていないが、定数 *Const* と変数 *Var* の字句の定義は Haskell と同じとする。ただし、`_` (アンダーバー) から始まる変数名はコンパイラ内部で使用するために予約済みとする。`while ~ do` 式や `begin ~ end` 式があるところが Haskell と異なり、命令型言語らしいところである。

### 6.1.2 字句解析・構文解析関数

次のような関数が既に定義されているものと仮定する。

```
myParse :: String -> Expr  -- 字句解析・構文解析の関数
```

- “`val x=2*2 in val y=x*x in y*y`” というソースプログラムは Expr 型のデータとして次のように構文解析される。

```
Val ("x", (App (App times (Const (TLit (Int 2))))
              (Const (TLit (Int 2))))))
      (Val ("y", (App (App times (Var "x")) (Var "x")))
          (App (App times (Var "y")) (Var "y"))))
```

ただし、`times` は `*` に対応する `Expr` の式である。

- “`\ f -> \ x -> f x`” という式は、

```
Lambda "f" (Lambda "x" (App (Var "f") (Var "x")))
```

というデータに構文解析される。

- `&&`, `||` は、それぞれ、

```
b1 && b2 ⇨ if b1 then b2 else False
```

```
b1 || b2 ⇨ if b1 then True else b2
```

という糖衣構文であるように構文解析されるようにしておく。

問 6.1.1 なぜ、`&&` や `||` はプリミティブ関数として定義すると良くないのか?

### 6.1.3 ターゲット言語

コンパイラーのターゲット言語である Haskell のサブセットの構文木を表現する型 `Target` 型を定義しておく。

ファイル `Target.hs`

```
1 type TDecl = (String, Target)
2 data Target = TLit Literal           -- 定数
3             | TVar String           -- 変数
4             | TIf Target Target Target -- if文
5             | TLet [TDecl] Target   -- let式(変数・関数定義)
6             | TLambda1 String Target -- ラムダ式
7             | TApp1 Target Target   -- 関数適用
8             | TUnit Target          -- return に相当
9             | TBind Target Target   -- (>=>) に相当
10            deriving (Show,Eq)
11 data Literal = Str String | Int Integer | Frac Rational
12              | Char Char           deriving (Show,Eq)
```

Util のコンパイラーとは次のような型を持つ関数である。

```
comp :: Expr -> Target -- コンパイラー
```

### 6.1.4 抽象構文と具象構文

ところで、上の Util の構文規則は \_\_\_\_\_ (空欄 6.1.3) (ambiguous) である。通常は曖昧さを避けるために、



comp 関数を理解するために、変換前と変換後を代数的データ型ではなく、それぞれ Util と Haskell の文法で記述したのが次の表である。(詳細はソースプログラムを参照すること。)

この表のなかでソース中で *Italic* フォントで示されている  $m, n$  などは任意の Util の式で、ターゲット中で  $\dot{m}, \dot{n}$  のように ` (ドット) が付いている式は、その comp による変換後の Haskell の式を表す。なお、delay については内部的に使用されるので、実際には Util のソースプログラムに現れることはない。

ソース ( Util )	ターゲット ( Haskell )
<code>c (ただし c は定数)</code>	<code>return c</code>
<code>x (ただし x は変数)</code>	<code>return x</code>
<code>val x = m in n</code>	<code><math>\dot{m}</math> &gt;&gt;= \ x -&gt; <math>\dot{n}</math></code>
<code>let f = \ x -&gt; m g = \ y -&gt; n in n</code>	<code>let f = \ x -&gt; <math>\dot{m}</math> x = \ y -&gt; <math>\dot{n}</math> in <math>\dot{n}</math></code>
<code>fa</code>	<code><math>\dot{f}</math> &gt;&gt;= \ _g -&gt; <math>\dot{a}</math> &gt;&gt;= \ _x -&gt; _g _x</code>
<code>\ x -&gt; m</code>	<code>return (\ x -&gt; <math>\dot{m}</math>)</code>
<code>delay m</code>	<code>return <math>\dot{m}</math></code>
<code>if c then t else e</code>	<code><math>\dot{c}</math> &gt;&gt;= \ _b -&gt; if _b then <math>\dot{t}</math> else <math>\dot{e}</math></code>
<code>while c do t</code>	<code>let _while = <math>\dot{c}</math> &gt;&gt;= \ _b -&gt; if _b then <math>\dot{t}</math> &gt;&gt;= \ _ -&gt; _while else () in _while</code>
<code>begin s; t; u end</code>	<code><math>\dot{s}</math> &gt;&gt;= \ _ -&gt; <math>\dot{t}</math> &gt;&gt;= \ _ -&gt; <math>\dot{u}</math></code>

要点は、変数や定数の出現など“副作用”が発生しないところには return が付くこと、関数適用は (>>=) を使った式に翻訳されることなどである。

また、Util プログラム中の +, -, \* などの二項演算子は、他の関数が return (\ x -> ...) という形に変換されること、戻り値はアクションを持たないことから、同名の Haskell 内のオペレーターを用いて、それぞれ

```
return (\ x -> return (\ y -> return (x+y)))
return (\ x -> return (\ y -> return (x-y)))
return (\ x -> return (\ y -> return (x*y)))
```

という Haskell の式に置換するようしておく。すると、comp 関数による他の部分の変換と整合する。

ソース ( Util )	ターゲット ( Haskell )
<code>⊗ (ただし ⊗ は二項演算子)</code>	<code>return (\ x -&gt; return (\ y -&gt; return (x ⊗ y)))</code>

この `comp` を用いて、例えば次の Util プログラムを変換<sup>2</sup>すると

```
fact = \ n -> if n==0 then 1 else n*fact (n-1)
```

次のような Haskell のプログラムが得られる。

```
1 fact = \ n ->
2   ((return (\ x -> return (\ y -> return (x == y))) >>=
3     \ _f -> return n >>= \ _x -> _f _x)
4     >>= \ _f -> return 0 >>= \ _x -> _f _x)
5     >>=
6     \ _b ->
7       if _b then return 1 else
8         (return (\ x -> return (\ y -> return (x * y))) >>=
9           \ _f -> return n >>= \ _x -> _f _x)
10          >>=
11          \ _f ->
12            (return fact >>=
13              \ _f ->
14                ((return (\ x -> return (\ y -> return (x - y)))
15                  >>= \ _f -> return n
16                    >>= \ _x -> _f _x)
17                  >>= \ _f -> return 1
18                    >>= \ _x -> _f _x)
19                  >>= \ _x -> _f _x)
20                >>= \ _x -> _f _x)
```

これは多くの冗長な部分を含んでいるので、前述の `monad law` などを利用して単純化すると、多くの `return` や `(>>=)` が消えて、次のような Haskell の式が得られる。

```
1 fact = \ n -> if n == 0 then return 1 else
2           fact (n - 1) >>= \ _x ->
3           return (n * _x)
```

## 6.2 最初のバージョン - Util1

最初のバージョン Util1 では、モナドはトリビアルな計算(何もしない計算)としておく。つまり、Util1 は副作用を持たない言語である。

ファイル Id.hs

```
1 newtype I a = I a
2
3 instance Monad I where
4   return a = I a
5   (I m) >>= k = k m
```

ここで、                     (空欄 6.2.1) は、Haskell の新しい型の宣言の形式の一つである。data 宣言と似ているが、フィールドが一つの構成子を一つしか持つことがで

<sup>2</sup>ただし、この `fact` 関数は副作用を含んでいないので、この変換自体にはあまり意味はない。

きない。data 宣言の構成子と異なり、newtype 宣言で導入される構成子は型変換の意味しか持たず、実行時の計算を伴わない。また、型の別名を宣言する type 宣言との違いは、型の変換が構成子によって明示的になる点である。(型クラスのインスタンスには、type 宣言で導入された型の別名は指定できない。例えばリストに通常の辞書式順序と異なる順序 (Ord のインスタンス宣言) を与えたいときは newtype を使って別名を用意し、別名に対してインスタンス宣言する必要がある)

上記の newtype 宣言では型構成子と構成子に同じ I という名前を使っているが、文脈でどちらか判断することができるので問題ない。

このとき、

```
fact = \ n -> if n==0 then 1 else n*fact (n-1)
```

という Util プログラムをコンパイルして実行する (unI (fact 9)) と、同じプログラムを Haskell として実行したときと全く同じ 362880 という値になる。

ただし、unI は次のように定義された型変換関数である。

ファイル Id.hs

```
1 unI :: I a -> a
2 unI (I a) = a
```

### 6.3 UtilST – 状態の導入

Util に更新 (代入) 可能な状態の概念を導入する。C 言語や Java 言語のように、変数に対して代入を導入することも可能であるが、非本質的な部分が多くなってしまうので、ここで紹介する例では、2 つだけ更新可能な “参照”  $x_P$  と  $y_P$  を導入することにする。2 という数は別に本質的なものではなく、いくつにすることも可能である。参照は set で値を代入し、get で値を取り出すことができる。

例えば、

```
begin set xP 1; set xP (get xP+3); get xP end
```

という UtilST プログラムを評価すると、(空欄 6.3.1) という結果が得られる。

状態を導入するために、やはり “計算の型” を定義する必要がある。まず 次の ST を定義する。

ファイル ST.hs

```
1 newtype ST s a = ST (s -> (a,s))
2
3 unST :: ST s a -> s -> (a,s)
4 unST (ST s) = s
5
6 instance Monad (ST s) where
7   return a = ST (_____ )
8   (ST m) >>= k = ST (\ s0 -> let { (a,m1) = m s0 }
9                               in unST (k a) s1)
```

```

10 -- ST, unST がなければ、次のようになる
11 -- m >>= k = \ s0 -> let { (a,s1) = m s0 } in k a s1

```

return a は状態 (s) の変更を行わず、a をそのまま返す計算である。このモナドの m >>= k は、m で変更された状態 (s1) をそのまま、k に受渡す計算である。参照は次のように定義する。

ファイル MyState.hs

```

1  type Pos s a = s -> (a, a -> s)
2
3  xP :: Pos (x,y) x
4  xP = \ (x,y) -> (x, \ x1 -> (x1,y))
5
6  yP :: Pos (x,y) y
7  yP = \ (x,y) -> (y, \ y1 -> (x,y1))

```

xP (yP) はペアの第 1 (第 2) 成分にアクセスするための参照である。参照に対する読み出し・書き込みのメソッドは、後で別のモナドでも使用するので、型クラス MyState のメソッドとして定義しておくことにする。

ファイル MyState.hs

```

1  class MyState m where
2    get :: Pos s a -> m s a
3    set :: Pos s a -> a -> m s ()

```

ST をこの MyState クラスのインスタンスとして宣言する。

ファイル ST.hs

```

1  instance MyState ST where
2    get p = ST (\ s -> (fst (p s), s))
3    set p v = ST (\ s -> ((), snd (p s) v))
4
5  -- 例えば get xP ≡ ST (\ (x,y) -> (x,(x,y)))

```

set は状態を書き換え、また get は状態の値の一部を複製している。

UtilST の set, get, ... という関数は、Haskell の set, get, ... にコンパイルされるようにしておく

ソース (Util)	ターゲット (Haskell)
set p m	p >>= \ _p -> m >>= \ _x -> set _p _x
get p	p >>= \ _p -> get _p

UtilST プログラム (右は対応する C プログラム):



```

1 fact = \ n -> begin
2   set xP 1; set yP n;
3   while get yP > 0 do begin
4     set xP (get xP * get yP);
5     set yP (get yP - 1)
6   end;
7   get xP
8 end

```

```

1 int fact(int y) {
2   int x = 1;
3   while (y > 0) {
4     x = x * y;
5     y = y - 1;
6   }
7   return x;
8 }

```

をコンパイルすると次のような Haskell の関数 (一部、見易くするために変数名の変更などを行っている) になる。

```

1 fact n = set xP 1 >>= \ _ ->
2   set yP n >>= \ _ ->
3   (let _while = get yP >>= \ y ->
4     if y > 0 then
5       get xP >>= \ x ->
6       get yP >>= \ y ->
7       set xP (x*y) >>= \ _ ->
8       get yP >>= \ y ->
9       set yP (y-1) >>= \ _ ->
10      _while
11     else return ()
12   in _while) >>= \ _ ->
13   get xP

```

fact 9 を実行する (fst (unST (fact 9) (0,0))) とその結果は 362880 になる。

階乗の場合、普通に関数的な定義のほうが簡潔だが、パラメーターの数が多い場合などは、このような命令的な書き方が簡潔になる場合もありうる。

この ST の定義では、エラー処理を考慮していない。エラー処理を行なうためには、この ST と (後述する) Maybe の定義を合成する必要がある。参考までに、次のようなモナドになる。

ファイル EST.hs

```

1 newtype EST s a = EST (s -> Maybe (a,s))
2
3 unEST :: EST s a -> s -> Maybe (a,s)
4 unEST (EST m) = m
5
6 instance Monad (EST s) where
7   return a = EST (\ s -> return (a,s))
8   (EST m) >>= k = EST (\ s0 -> case m s0 of
9     Just (a,s1) -> unEST (k a) s1
10    Nothing      -> Nothing)

```

問 6.3.1 次の C の関数とほぼ同等な Haskell の関数をモナドを用いて作成せよ。

```
1. int foo(int n) {
    int i = 1, j = 1;
    while (i < n) {
        i = i + j;
        j = i - j;
    }
    return i;
}
```

## 6.4 UtilIO – 入出力の導入

入出力は、入出力ストリームを状態の一種と考えれば、6.3 節の UtilST と同じ方法で取り扱うことができる。

計算のモナドの定義は 6.3 節と基本的に同じだが、状態に入力と出力のストリームを表す String 型の部分を追加しておく。

ファイル MyStream.hs

```
1 type WithIO s = (s,String,String)
```

ファイル MyIO.hs

```
1 newtype MyIO s a = MyIO (WithIO s -> (a, WithIO s))
2
3 unMyIO :: MyIO s a -> WithIO s -> (a, WithIO s)
4 unMyIO (MyIO m) = m
```

参照のプリミティブの定義は次のようになる。

ファイル MyIO.hs

```
1 instance MyState MyIO
2 get p = MyIO (\ (s,i,o) -> (fst (p s), (s,i,o)))
3 set p v = MyIO (\ (s,i,o) -> ((), (snd (p s) v,i,o)))
```

入出力に関するプリミティブも後で別のモナドで使用するの、型クラス MyStream のメソッドとして定義しておく。

ファイル MyStream.hs

```
1 class MyStream m where
2 readChar :: m Char
3 eof :: m Bool
4 writeStr :: String -> m ()
```

ファイル MyIO.hs

```
1 instance MyStream (MyIO s) where
2 readChar = MyIO (_____)
3 eof = MyIO (\ (s,i,o) -> (null i, (s,i,o)))
4 writeStr v = MyIO (_____)
```

readChar は入力ストリームから 1 文字を取り出し、writeStr str は出力ストリーム o に str を追加している<sup>3</sup>。write という関数を次のように定義しておく。

ファイル MyStream.hs

```
1 write :: (Show v, MyStream m) => v -> m ()
2 write v = writeStr (show v)
```

すると、次の UtilIO プログラム(ただし、// は整数の除算を表す演算子とする。)

```
1 foo = \ n -> begin
2   set xP n;
3   while get xP > 0 do begin
4     write (get xP % 10);
5     set xP (get xP // 10)
6   end
7 end
```

をコンパイルした結果は、

```
1 foo n = set xP n >>= \ _ ->
2   let _while
3     = get xP >>= \ _x ->
4       if _x > 0 then
5         get xP >>= \ _x ->
6         write (_x 'mod' 10) >>= \ _ ->
7         get xP >>= \ _x ->
8         set xP (_x 'div' 10) >>= \ _ ->
9         _while
10      else return ()
11   in _while
```

となり、これを

```
let (_,(_,_,o)) = unMyIO (foo 12345) ((0,0), "", "") in o
```

のように実行したときの出力は、"54321" になる。

問 6.4.1 次の C の関数とほぼ同等な Haskell の関数をモナドを用いて作成せよ。

```
1.
int bar(int n) {
  int i, j;
  for (i = 0; i < n; i++) {
    for (j = 0; j <= i; j++) {
      printf("*");
    }
    printf("\n");
  }
  return i;
}
```

<sup>3</sup>++の計算量は左オペランドの長さに比例するので、この定義のように文字列の後ろに新しい文字列を追加(++)していくと、出力文字列が長くなるにしたがって効率が悪くなる。これを避けて効率の良い定義を与えることも可能であるが、ここでは簡単のために++を使った定義を採用する。

## 6.5 UtilErr – エラー処理の導入

次に Util にエラー処理を導入する。UtilErr は、生真面目に (?) エラー処理を行ない、部分式にエラーがあれば式全体もエラーになるようにする。この場合、UtilErr は ( Haskell のような ) 遅延評価ではなくて、関数の引数を必ず先に評価する \_\_\_\_\_ (空欄 6.5.1) (eager evaluation) を模倣することに注意する必要がある。

エラーと正常な振舞いを区別するために、次のような標準ライブラリに用意されているデータ型 Maybe を使用する。

```
-- Prelude に定義済み
data Maybe a = _____ | _____ deriving (Show,Eq)
```

正常な振舞いは Just という構成子で表す。エラーの場合は Nothing という構成子を用いる。この型に対して次のようなインスタンス宣言がされている。

```
1 -- Prelude に宣言済み
2 instance Monad Maybe where
3   return a = Just a
4   (Just a) >>= k = _____
5   Nothing >>= k = _____
```

このモナドの  $m \gg= k$  は、まず  $m$  を計算し、その計算が正常終了すれば、その値を  $k$  という関数に渡す。しかし、いったん  $m$  でエラーが起こると、 $k$  は評価されず、\_\_\_\_\_ (空欄 6.5.2) していくことを表している。

さらに、MonadPlus というクラスに属するいくつかのメソッドを用意する。mzero は \_\_\_\_\_ (空欄 6.5.3) 状況をつくり出すときに用いる。

```
1 -- モジュール Control.Monad に定義済み
2 class Monad m => MonadPlus m where
3   mzero :: m a
4   mplus :: m a -> m a -> m a
5
6 -- モジュール Control.Monad に宣言済み
7 instance MonadPlus Maybe where
8   mzero = Nothing
9   ... -- mplus の定義は後掲
```

“プリミティブ関数” もエラー処理を利用するように書き換えることができる。例えば、割り算 ( / ) は次のような Haskell の式に置換されるようにする。

```
\ x -> return (\ y -> if y==0 then mzero
                    else return (x/y))
```

これで 0 で割ろうとした場合にはエラーが報告される。

例えば

```
(\ x -> 0) (1/0)
```

のような式は、Haskell では  $1/0$  の部分式は \_\_\_\_\_ (空欄 6.5.4) に全体の結果が 0 となるが、UtilErr では次のような Haskell プログラムに翻訳され、

```

1 (if 0 == 0 then mzero
2   else return (1/0)) >>= \ _x ->
3   return 0

```

実行すると(エラーが起こったことを表す) \_\_\_\_\_ (空欄 6.5.5)という結果になる。

## 6.6 例外処理の導入

例外のモナド Maybe を利用して、Java の try ~ catch のように例外を捕捉する構文を導入することも可能である。

Util の BNF には以下の構文を追加する。

$$Expr \rightarrow \dots \mid \mathbf{try} \ Expr \ \mathbf{catch} \ Expr$$

“**try** *m* **catch** *h*” は *m* を評価し、エラーがなかった場合は、その戻り値を **try** 式の戻り値とする。しかし *m* の評価中にエラーが生じた場合は、*h* を評価する。Util の “**try** *m* **catch** *h*” は “*m*” ‘mplus’ *h*” という式として構文解析されるようにしておく。

また、fail という Util の関数は、Haskell の mzero を返す関数にコンパイルされるようにしておく。この関数は、Java の throw 文に対応する。

ソース ( Util )	ターゲット ( Haskell )
<b>try</b> <i>m</i> <b>catch</b> <i>n</i>	<i>m</i> ‘mplus’ <i>n</i>
fail ()	mzero

Maybe に対する mplus は第 1 引数を実評価し、\_\_\_\_\_ (空欄 6.6.1) 第 2 引数を実評価する関数である。

```

1 instance MonadPlus Maybe where
2   ...      -- mzero は紹介済
3   Just a  ‘mplus’ _ = Just a
4   Nothing ‘mplus’ m = m

```

例えば

```
try 1/0 catch 99999
```

という UtilErr プログラムをコンパイルすれば、出力される Haskell プログラムは、

```

1 (if 0 == 0 then mzero else return (1/0))
2 ‘mplus’ (return 99999)

```

であり、その結果は Just 99999.0 である。



```

1 test1 = (return 1 'mplus' return 2) >>= \ x ->
2         (return 0 'mplus' return 4) >>= \ y ->
3         if y == 0 then mzero else return (x/y)

```

test1は、[0.25,0.5]となる。失敗している計算については結果に現れていないことに注意する。同じプログラムを UtilErr でコンパイルすると、UtilErr ではバックトラッキングが起こらないので、この計算は全体が失敗 (Nothing) に終わる。

なお、次の head を用いてリストの頭部を取るにより、成功した最初の計算だけを返すことも可能である。

```

1 -- Prelude に定義済み
2 head :: [a] -> a
3 head (x:_) = x

```

head test1 の値は 0.25 となる。この場合、Haskell が \_\_\_\_\_ (空欄 6.7.3) を採用しているため、他の選択肢の計算は行なわれない。そのため選択肢が無数あるような場合でも最初の選択肢の計算結果を出力することができる。

問 6.7.1 非決定性と状態の両方の特徴を持つ計算の型として、

```

1 newtype STL s a = STL (s -> ([a],s))
2 newtype LST s a = LST (s -> [(a,s)])

```

の2つのバリエーションが考えられる。このそれぞれに対して、コンパイラーの定義を完成させ、2つの違いを説明せよ。

## 6.8 さらに詳しく知りたい人のために ...

Parsec [1] はモナドを利用した有名なパーサーライブラリーである。[3] にも、モナドを用いてパーサーを構築する技法の解説がある。[2] はモナドを用いてインタプリタを構築する方法を解説している。[4] は、Prolog のカット等のオペレーターの意味を整理している。

### この章の参考文献

- [1] Daan Leijen and Erik Meijer 「Parsec: Direct Style Monadic Parser Combinators for the Real World」  
 Technical Report UU-CS-2001-35, Dept. of Comp. Sci, Universiteit Utrecht, 2001 年, <http://www.cs.uu.nl/people/daan/parsec.html>

- [2] Philip Wadler 「 The essence of functional programming 」  
19th Annual Symposium on Principles of Programming Languages (invited talk),  
1992 年 1 月
- [3] Philip Wadler 「 Monads for functional programming 」  
Program Design Calculi, Proceedings of the Marktoberdorf Summer School, 1992  
年 7-8 月
- [4] Ralf Hinze 「 Prological Features in a Functional Setting Axioms and Implementations 」  
Third Fuji International Symposium on Functional and Logic Programming, 1998  
年